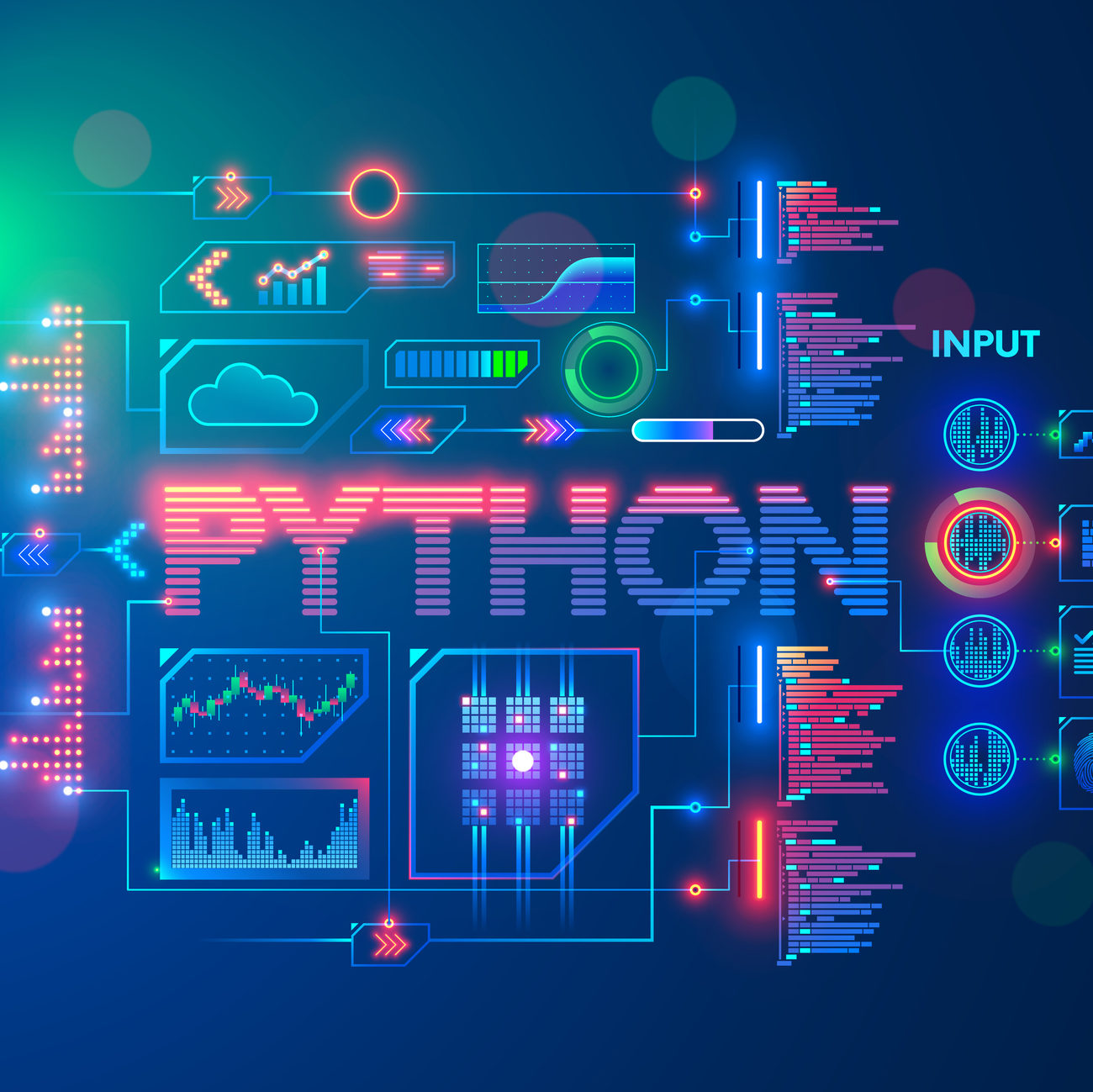
**Assignment 2**



**Submitted by:  
Name: Nikitha Donthi  
Student Id: 100953192  
Date: 15/11/2024**

**Part 1: Identify the two Problem**

**Problem 1: Automated Daily Task Reminder**

**Problem Statement:**

Some tasks will have to be managed daily to help in keeping organized in general. Many meetings, deadlines, and tasks juggled all at once can be hard to keep track of; a reminder system can keep you from forgetting important activities. Automating such a process not only saves time but raises productivity with timely reminders.

**Solution Overview:**

In this problem, Python is used to develop an automated system for reminding daily tasks. The system sends an email automatically at a specific time each day to remind users to check the to-do list or to go through any urgency related to performing a daily task.

**Libraries Used:**

**1. smtplib:**

• This built-in library is used to send emails via an SMTP (Simple Mail Transfer Protocol) server. In this case, it connects to Gmail's SMTP server.

• It allows personalization of the email by embedding a task reminder message, subject line, and recipient.

**2. schedule:**

• A light-weight job scheduling library for Python. This allows the programmer to run Python functions periodically, something which we want to do by setting specific times for different things to remind us.

• This is useful because it allows the scheduling of the reminder email at specific times (e.g., 8:00 AM daily) using just one line of code.

**3. time:**

• This is a standard Python library used to create a pause in the program so that it can check and execute tasks at the specified time intervals.

**About the Code**

**Step 1: The function send\_reminder() can be defined as**

This function sends a request to Gmail's SMTP server through smtplib.SMTP() to connect to your e-mail account by logging in with your credentials and finally sends an e-mail to the recipient with the task reminder message.

**Step 2: Scheduling the reminder:**

It allows for scheduling the function send\_reminder() every day at 8:00 AM using schedule.every().day.at("08:00").do(send\_reminder).

**Step 3: Run Continuously:**

The while True loop now ensures that the script runs constantly, pending the time to send the email. It checks every 60 seconds for any pending tasks and performs them.

**Practical Use and Benefits:**

1. **Productivity:** Automating reminders helps the user remember things without having to look personally into their to-do list to make sure that everything is found; this therefore boosts productivity.
2. **Consistency:** It is a reminder sent at the same time every day without fail so that one can keep consistent with daily tasks.
3. **Simple yet Effective:** This solution doesn’t require a complex setup or any third-party applications—just a simple Python script running on the user's machine.

**Future Improvements:**

1. **Multiple Reminders:** Allow the user to add multiple reminders in a day-say, morning, afternoon, and evening reminder-and be able to set a different message for each one.
2. **GRAPHICAL USER INTERFACE (GUI):** The program can have a GUI to allow the end user to set up easily the reminder time, frequency, and task message without having to touch the Python code.
3. **Calendar Application Integration:** The reminder system can be extended to integrate with Google Calendar or another productivity application, pull events, and remind the user as to when appointments are scheduled.

**Problem 2: Extracting and Visualizing Website Data**

**Problem Statement:**

The internet has a quite vast amount of data lying around, exclusively useful for business and any form of researchers. Web scraping is, therefore, able to extract relevant data from websites for analysis. After collecting the data, visualizing the data allows sense to be obtained from large volumes of data by spotting trends. We'll scrape an ecommerce website for some basic content like product titles and prices, then visualize the price trends.

**Solution Overview:**

Web scraping is all about fetching data from the HTML structure of a website, and the visualization of the extracted data highlights the meaningful pattern. Here, this problem deals with scraping product data from a sample e-commerce website and saving them in CSV format for further analysis.

**Libraries Used:**

1. **requests:** This library fetches the HTML content of the webpage by sending an HTTP request. Later, this content is parsed and analyzed for meaningful data extraction.
2. **BeautifulSoup:** This is a Python library that can be used for parsing HTML and XML documents. It helps to identify specific tags within the HTML structure that contain the data we want to extract.
3. **csv:** Following the extraction, this module will help save data extracted into a CSV file. A CSV file is one of the more general means of storing structured data and can easily be imported into spreadsheet tools like Excel.
4. **matplotlib optional:** This library is used in graphing data. For the solution of this problem, it shall be used to plot charts showing time series of products' prices.

**Code Breakdown:**

**Step 1: Fetch Website HTML:**

The script fetches the HTML of the provided URL using requests.get(). It then checks if the request was successful with response.raise\_for\_status().

**Step 2: Parse HTML and Extract Data:**

BeautifulSoup is employed to parse the HTML, while the relevant tags of product titles and prices are tugged out using find\_all() and find(), respectively. The extracted data from here is stored in a dictionary.

**Step 3: Saving Data to CSV:**

The information is then written to a CSV file by csv.DictWriter. For each product, a row is created in the CSV file containing its title and price.

**Practical Application and Benefits:**

1. **Market Research:** The same solution can be used to get all information on competitor prices, the trend of pricing for a certain product, or analyze customer reviews and other publicly available data.
2. **Data-driven decisions:** The extraction and visualization of data give a business insight into behavioral trends in the market, consumer preference, and price action.
3. **Automation:** Web scraping can be automated to periodically fetch updated data from websites, ensuring that the dataset is current at any moment without manual intervention.

**Challenges Faced:**

1. **Difference in HTML Structure:** Each website has a different HTML structure, and hence to find the exact tags that are to be used for scraping, each website needs to be inspected.
2. **Legal and Ethical Considerations:** First, it is always well and good to check a website's robot.txt file for any policy on its web scraping. Some websites may disallow auto-scraping, and such websites should be accorded their wishes.

**Future Improvements:**

1. **Handling Dynamic Content:** Over here, if any website uses JavaScript to load content, we could use some tool like Selenium or Puppeteer for handling the loading of dynamic content.
2. **Data Cleaning:** More Cleanup and normalization routines could be added, such as removing special characters from a product's price or converting them into numeric values.
3. **Data Visualization:** The price trend over a period can be visualized using Matplotlib or Seaborn to provide improved insight into the data, after scraping and cleaning.

**Part 3: Reflect on the Solution**

**Reflection on Extracting and Visualizing Website Data**

The task of Extracting and Visualizing Website Data was more challenging than I initially expected. Parsing data from dynamic websites using Selenium required understanding browser automation and handling JavaScript-heavy pages.

**Challenges:**

The most challenging part was overcoming restrictions like CAPTCHA or rate-limiting on certain websites. Setting up pandas for preprocessing large datasets was also tricky but manageable with practice.

**Learning and Resources:**

I learned a lot about web scraping techniques and the importance of adhering to ethical guidelines (e.g., checking a website’s robots.txt file). The visualization step was the most enjoyable, as it allowed me to creatively present data. I used a combination of matplotlib and seaborn to create bar graphs and line charts.

**Testing:**

I tested the scraper by extracting data from public websites like weather or stock data sites. The results were visualized successfully, and the program ran efficiently. However, handling larger datasets slowed the process, prompting me to explore optimization techniques.

**Enjoyment and Future Plans:**

This project was highly engaging as it combined automation, data analysis, and visualization. In the future, I plan to add more interactive elements to the visualization, perhaps by using Plotly for web-based dashboards.

**Final Thoughts**

Both projects Automated Daily Task Reminder and Extracting and Visualizing Website Data highlighted the versatility of Python. They provided a balance of challenges and learning opportunities. These solutions not only addressed practical problems but also opened doors to explore advanced topics like API integration and data analysis further.

**References**

1. Crummy. (n.d.). Beautiful Soup documentation. Retrieved November 19, 2024, from <https://beautiful-soup-4.readthedocs.io/>
2. Python Software Foundation. (n.d.). csv — CSV file reading and writing. Retrieved November 19, 2024, from <https://docs.python.org/3/library/csv.html>
3. Real Python. (n.d.). Web scraping with Beautiful Soup and Python. Retrieved November 19, 2024, from <https://realpython.com/beautiful-soup-web-scraper-python/>
4. Reitz, K., & Chisholm, T. (n.d.). Requests: HTTP for humans. Retrieved November 19, 2024, from <https://docs.python-requests.org/en/latest/>